Singularity Overview

Galen Hunt and James Larus
Microsoft Research

July 17, 2006
MSR Faculty Summit

Large, Diverse Research Team

- Lead by Galen Hunt and Jim Larus
- **MSR Cambridge**
  - Paul Barham, Richard Black, Tim Harris, Rebecca Isaacs, Dushyanth Narayanan
- **MSR Redmond**
  - Advanced Compiler Technology Group:
    - Juan Chen, Qunyan Mangus, Mark Plesko, Bjarne Steensgaard, David Tarditi
  - Foundations of Software Engineering Group:
    - Wolfgang Grieskamp
  - Operating Systems Group:
    - Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, Steven Levi
  - Security and Distributed Systems:
    - Dan Simon, Brian Zill
  - Software Design and Implementation Group:
    - John DeTreville, Ben Zorn
  - Software Improvement Group:
    - Manuel Fahndrich, James Larus, Sriram Rajamani, Jakob Rehof
- **MSR Silicon Valley**
  - Martin Abadi, Andrew Birrell, Ulfar Erlingsson, Roy Levin, Nick Murphy, Ted Wobber

“Modern” OS And Applications

“Modern” OS And Applications

- Design parameters
  - scarce resources
  - benign environment
  - knowledgeable and trained users

“Modern” OS And Applications

- Design parameters
  - scarce resources
  - benign environment
  - knowledgeable and trained users

World Changed

- Hardware and software industries were wildly successful
  - machines are fast
  - memory is cheap
  - computers are ubiquitous
- Malicious environment
  - ubiquitous worms, viruses, scams, attacks, …
- Few users understand computers or software

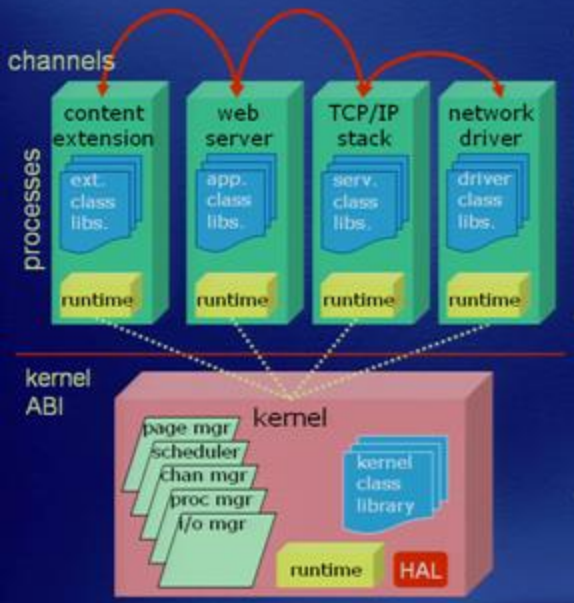
Singularity

- Goal: technology and techniques to build more dependable systems
- Dependable: predictable behavior and easily understood usage model
  - consumer satisfaction: new car vs. new PC
    - car has .99 to .999 availability (9-90 hours down time/yr)
- Research on new OS, languages, and tools
  - attack problem from multiple directions
  - working research prototype (not Windows replacement)
- No magic bullet
  - mutually reinforcing improvements to languages and compilers, systems, and tools

Key Approaches

1. Pervasive use of safe (& analyzable) programming languages
   - type safety and memory safety
   - including device drivers, OS components, applications
2. Improve system resilience despite software errors
   - failure boundaries between components
   - improve extension model
   - explicit error notification
3. Increased verification
   - specification at multiple levels of abstraction
   - closed environments with explicit cross-domain interfaces
   - design for verifiability

Singularity OS



- Closed Kernel
  - 95% written in C#
    - 17% of files contain unsafe C#
    - 5% of files contain x86 or C++
  - OS services & drivers in processes
  - kernel closed at boot time
- Software isolated processes (SIPs)
  - all user code is verified safe
  - some unsafe code in trusted runtime
  - processes closed at start time
- Safe and efficient communication via strong interfaces
  - channels between processes
  - channel behavior is specified & checked
  - checked behavior enables efficient communication
- Type safety is crux of verification and protection

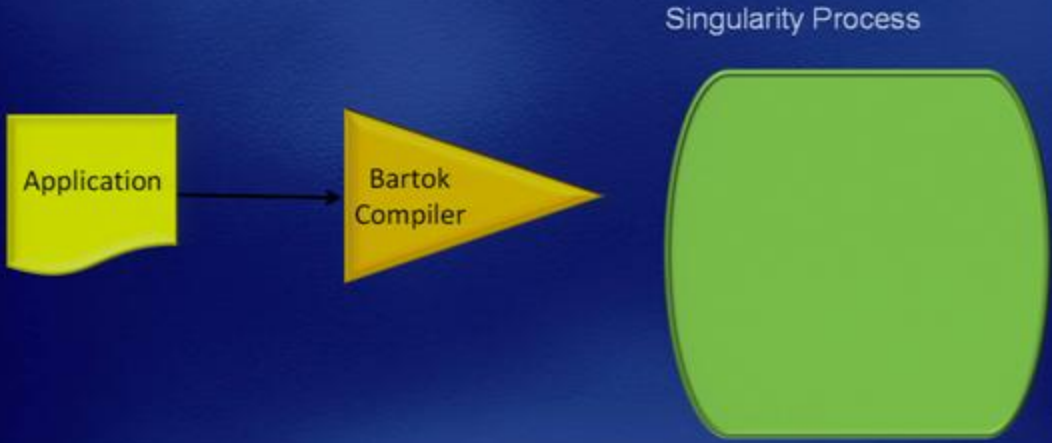
Challenge 1: Pervasive Safe Languages

- Singularity is written in extended C#
  - actually Spec#
    (C# + pre/post-conditions and invariants)
- Added features for systems programming
  - increase programmer control over allocation, initialization, and memory layout
- Language design to support programming and verification
  - message passing
  - factoring libraries into composable pieces
  - compile-time reflection

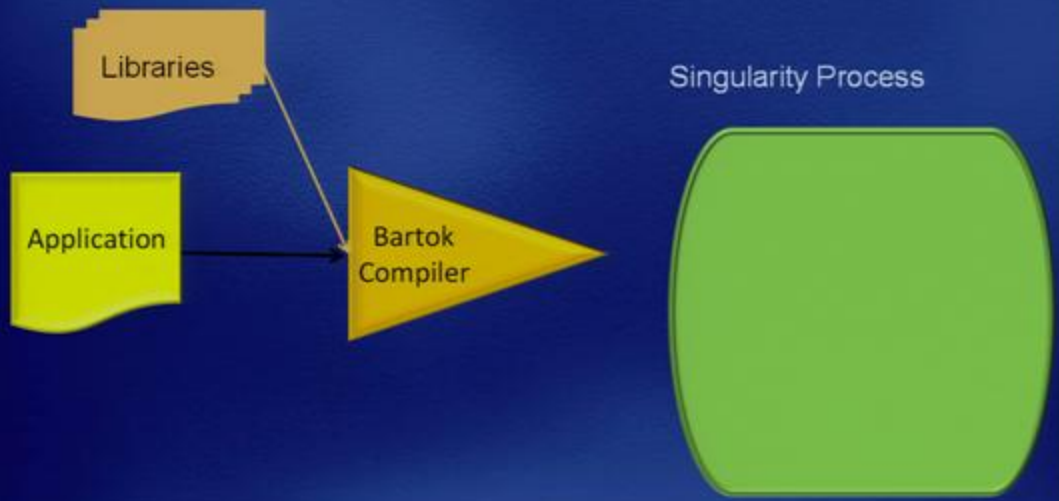
What About The Runtime?

- JVM & CLR's design not always appropriate
  - rich runtime ("one size fits all")
    - monolithic, general-purpose environment
    - large memory footprints (~4 MB process for CLR)
    - many dependencies (CLR PAL requires >300 Win32 APIs)
  - JIT compilation
    - increases runtime size and complexity
    - unpredictable performance
  - replicate OS functionality
    - security, threading, configuration, etc.
    - more is less

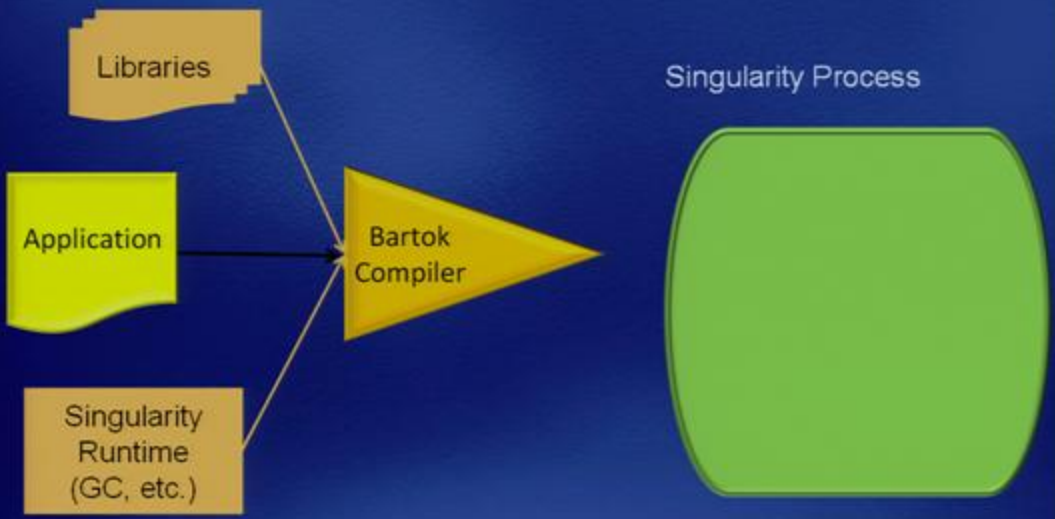
Singularity Runtime



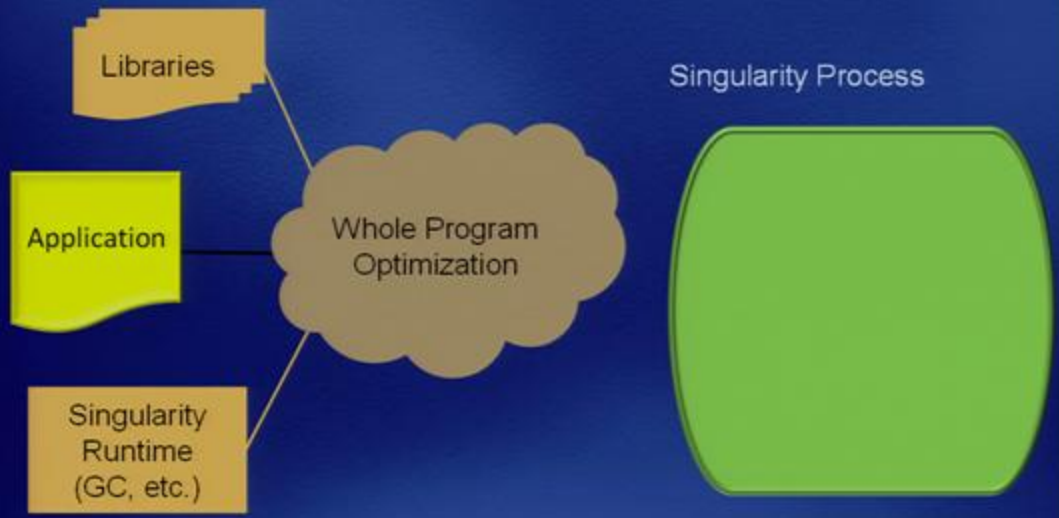
Singularity Runtime



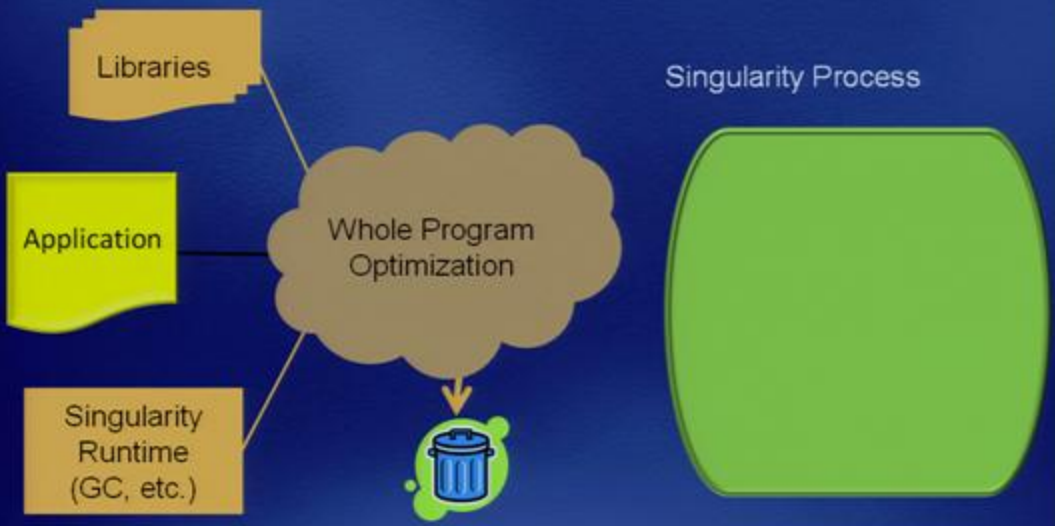
Singularity Runtime



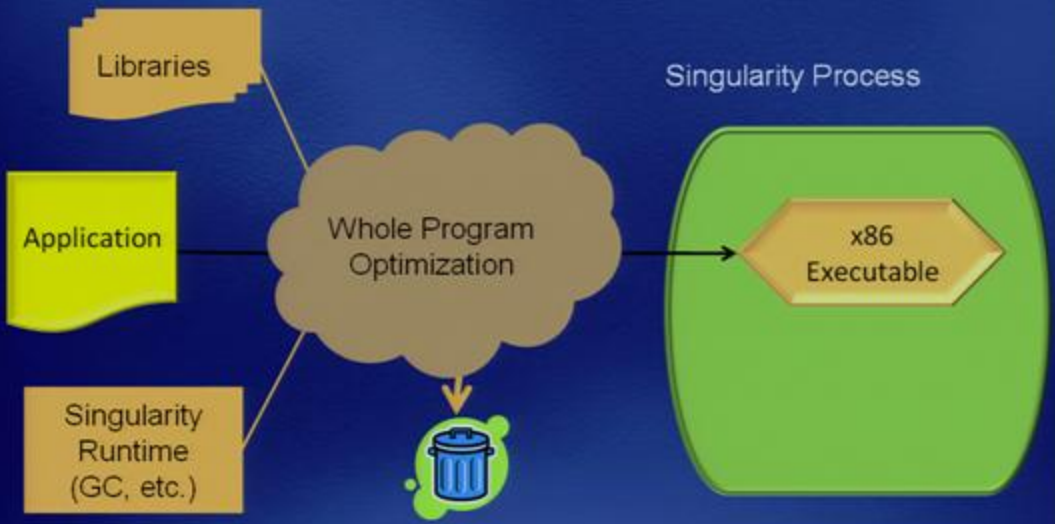
Singularity Runtime



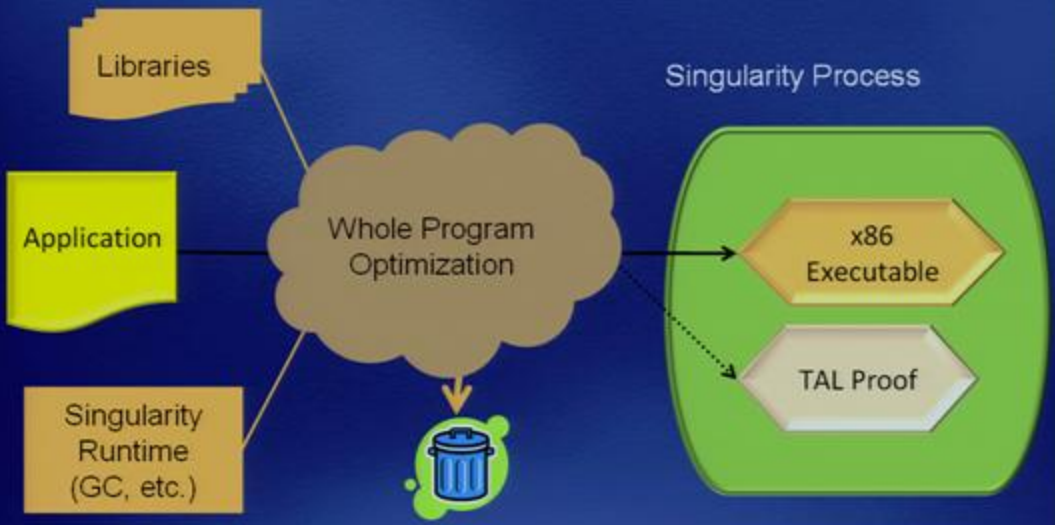
Singularity Runtime



Singularity Runtime



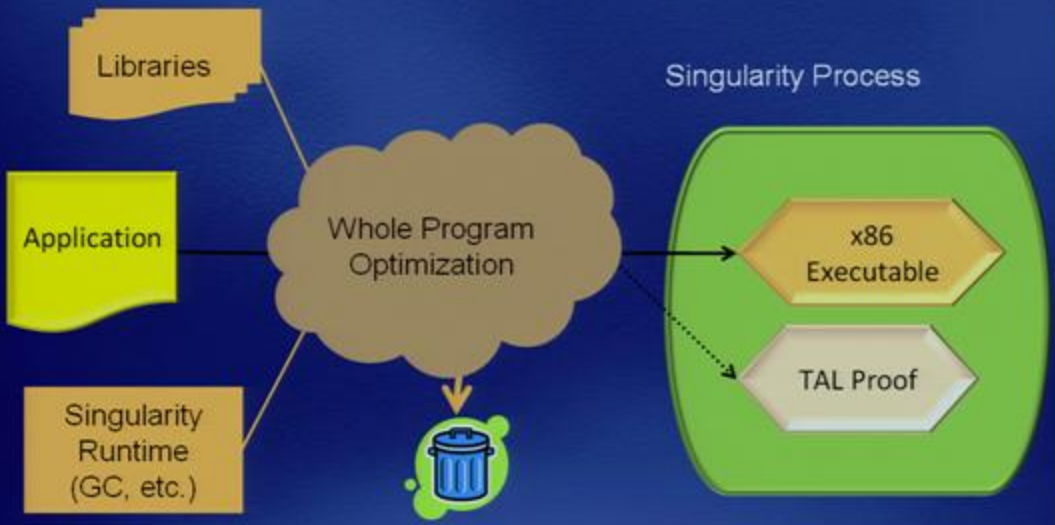
Singularity Runtime



Small, Customizable Runtime

- Small execution environment
  - ahead-of-time, global optimizing compiler (MSR Bartok) specializes runtime and libraries
    - eliminate code for unused/disabled language features and unused application/library code
  - factorable runtime and libraries
- Runtime, garbage collector, and libraries selectable on per-process basis
  - reduce memory and computation overhead
  - enforce design discipline and system policies per process
- Eliminate OS functionality from runtime
  - security, resource allocation, etc.
- Provide OS mechanism for enforcing system policy
  - runtime can constrain behavior (e.g. driver environment)

Singularity Runtime



Runtime Overhead

	Memory footprint "Hello World" process			
	Singularity	FreeBSD 5.3	Linux 2.6.11 (Red Hat FC4)	Windows XP (SP2)
C - static lib		232K	664K	544K
C++ - static lib		704K	1,216K	572K
C# - w/ GC	408K*			3,750K

- C# process w/ GC has similar memory footprint to C++
  - minimal process (no GC or exceptions) is ~16K

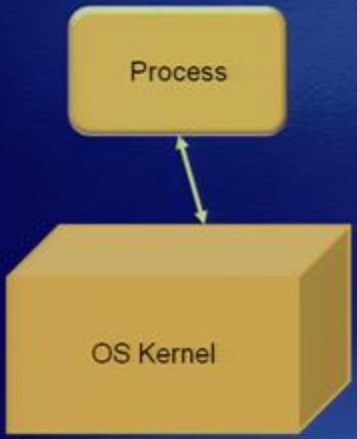
Challenge 2:

Run-Time Resilience

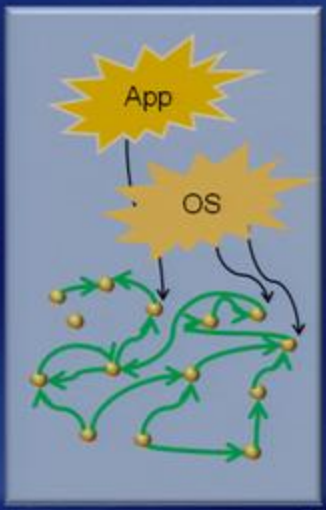
- Software errors should not cause system failure
- Resilient system architecture
  - isolate system components to prevent data corruption
  - provide clear failure notification
  - implement policy for restarting failed component

Process Architectures

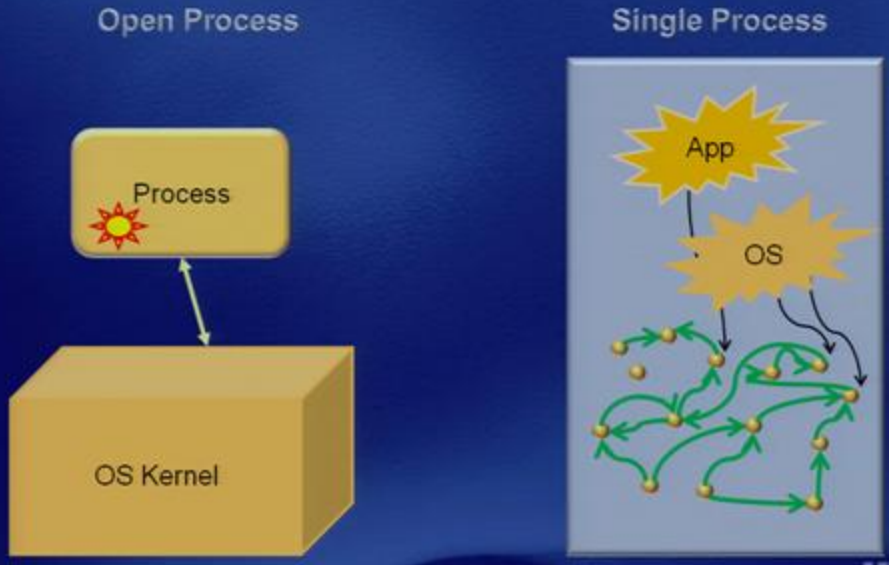
Open Process



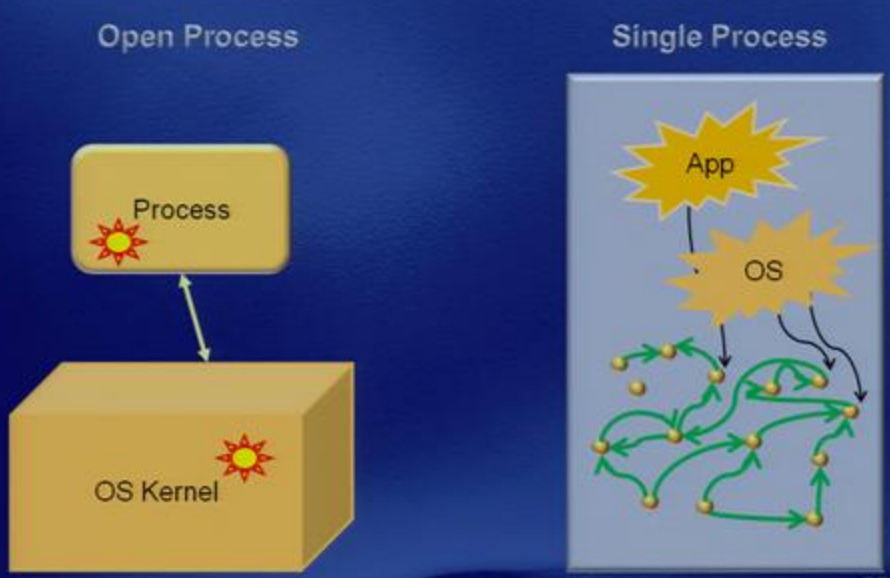
Single Process



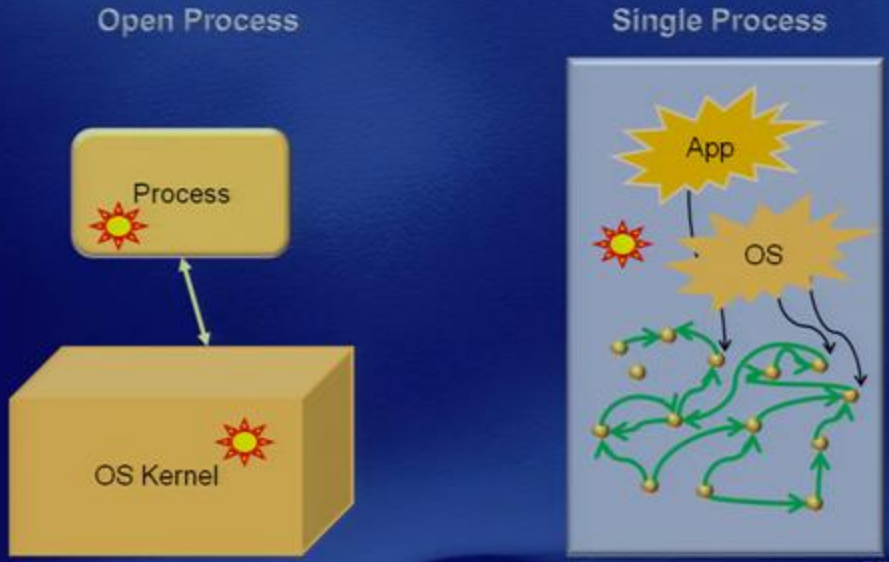
Process Architectures



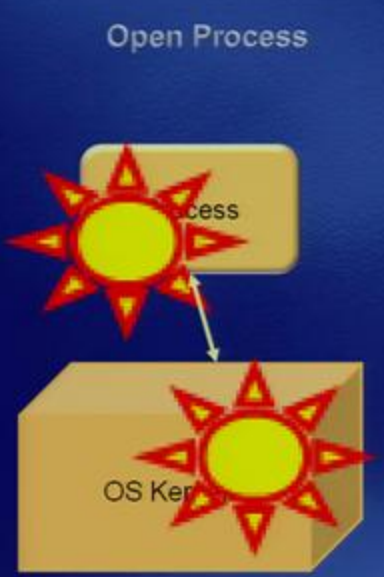
Process Architectures

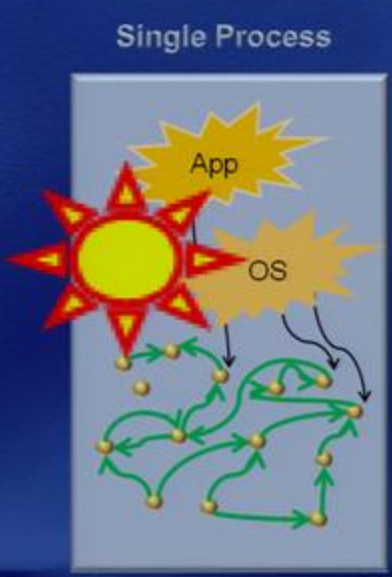


Process Architectures



Process Architectures





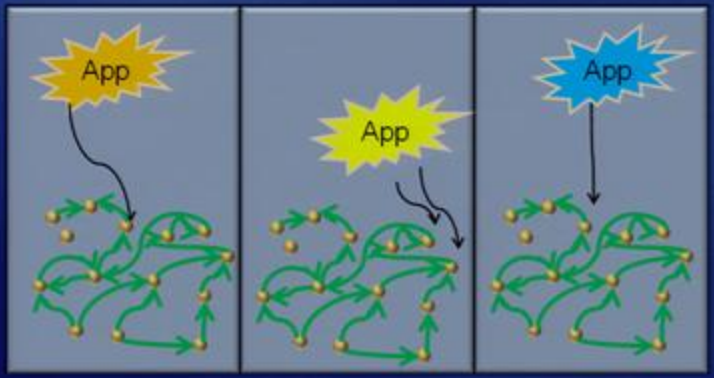
Open Process Architecture

- Open processes
  - dynamic code loading and runtime code generation
    - DLLs, Java class loading, browser plug-ins, device drivers in kernel, etc.
  - cross-process memory sharing
  - system API allows one process to alter state of another
- Near ubiquitous (Windows, Unix, etc.)
  - originated in Multics
- Shared state reduces dependability
  - 85% of Windows crashes are caused by third party code in kernel
  - interfaces between extension and host are often poorly documented and understood
  - no isolation boundary between code and extension
  - extension can access non-public interfaces (reflection)

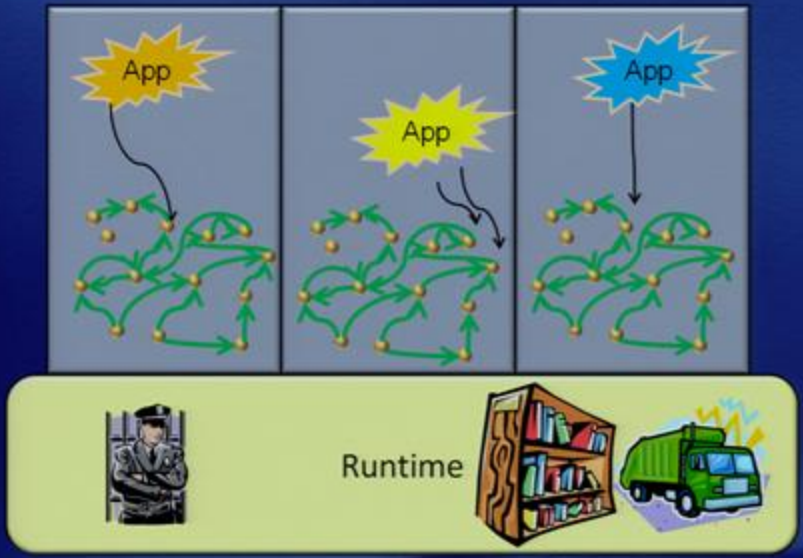
Single Process Architecture

- All code and data in single address space
  - rely on language and memory safety to isolate components
  - dynamic code loading and runtime code generation
  - easy data sharing
- Xerox PARC (Cedar, Smalltalk, etc.) and Lisp Machine model
  - Java and .NET model as well
- Runtime is single point of failure
  - shared runtime must also meet all applications' requirements
- Rely on garbage collection to reclaim resources
  - finalizers
- Difficult to constraint interactions

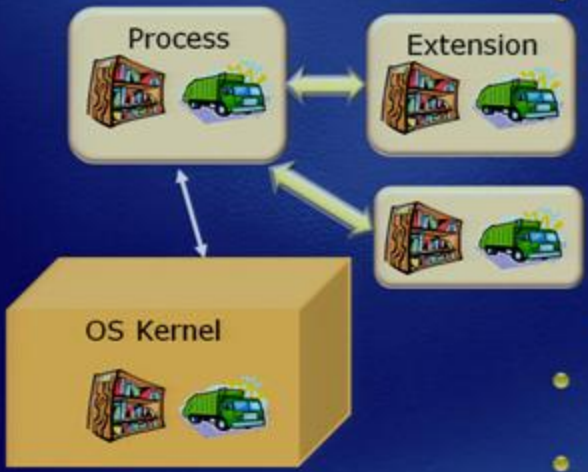
Isolates And AppDomains Are Still Interdependent



Isolates And AppDomains Are Still Interdependent



Singularity Sealed Processes



- Singularity processes are sealed
  - no dynamic code loading or run-time code generation
    - all code present when process starts execution
  - extensions execute in separate processes
    - separate closed environments with well-defined interfaces
  - no shared memory
- Process is fundamental unit of failure isolation
- Better: security, verification, failure handling, optimization

Static Benefit Of Sealed Processes

Program	Whole Code	Reachable Code	% Reduction
Kernel	2.37 MB	1.29 MB	46%
IDE Disk Driver	1.85 MB	455 KB	75%
Web Server	2.73 MB	765 KB	72%
Content Extension	2.14 MB	502 KB	77%

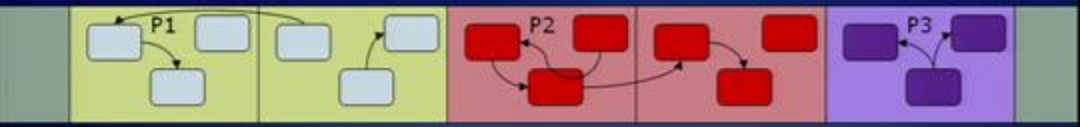
- Reduces process code size by up to 75%.
- Fewer code paths => better optimization & error analysis

Need For Lightweight Processes

- Existing processes rely on expensive hardware virtual memory and protection mechanisms
  - VM prevents reference to other processes' pages
  - protection prevents unprivileged code from access system resources (e.g. VM page tables)
- Processes are expensive to create and schedule
  - encourages monolithic program development
  - large, undifferentiated applications
  - dynamic code loading
  - threading to allow independent control flow

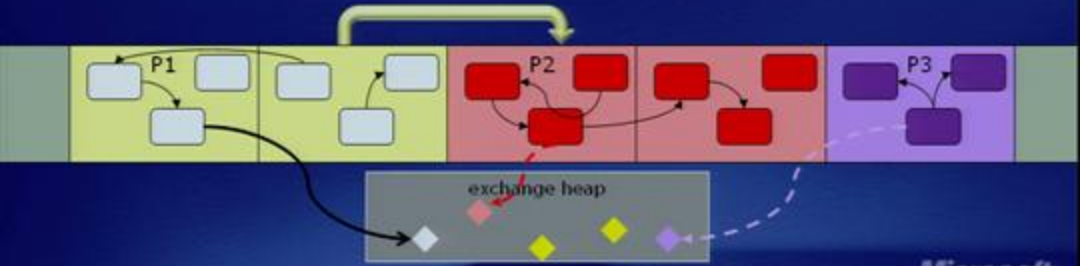
Software Isolated Processes (SIPs)

- Protection and isolation enforced by language safety and kernel API design, not hardware
  - process owns a set of pages
  - all of a process's objects reside on its pages (object space, not address space)
  - language safety ensures process can't create or mutate references to other pages
- Global invariants:
  - no process contains a pointer to another process's object space
  - no pointers from exchange heap into process



Interprocess Communications

- Channels are strongly typed (value and behavior), bidirectional communications ports
  - messages passing with extensive language support
- Messages live outside processes, in exchange heap
  - only a single reference to a message
- "Mailbox" semantics enforced by linear types

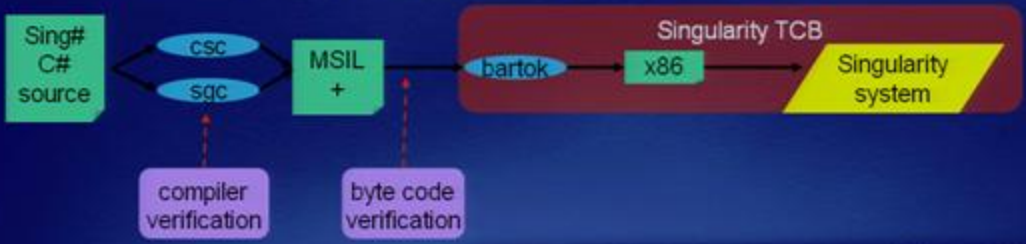


Failure Isolation

- SIPs are failure containers
  - no shared implementation or state across SIPs
  - process runtimes are distinct
- On SIP failure:
  - clean failure notification on peer channel endpoints
  - resources reclaimed by OS
- Recovery feasible, not automatic or transparent
  - peers can recover and continue

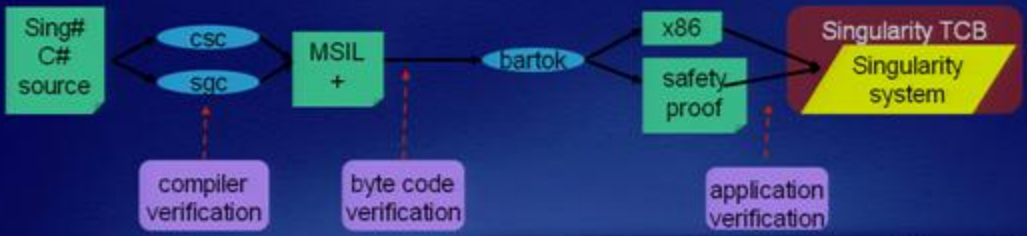
Would You Trust Your System To A Type System?

- Process integrity depends on type and memory safety
  - currently trust compiler and runtime
- TAL can eliminate compiler from trusted computing base
- Working on verifying the GC as well

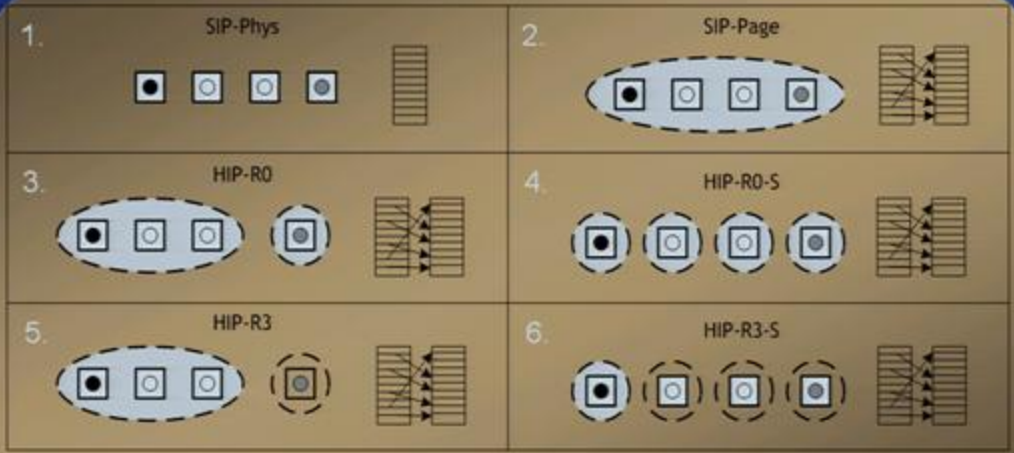


Would You Trust Your System To A Type System?

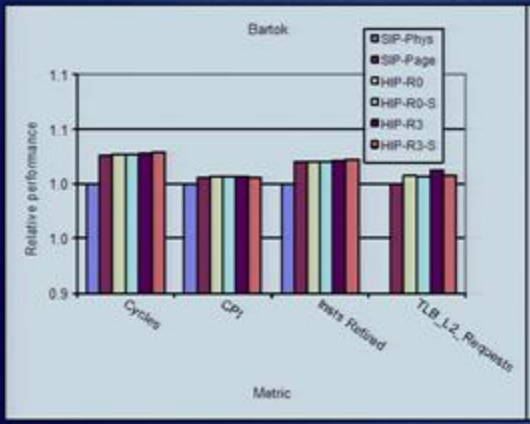
- Process integrity depends on type and memory safety
  - currently trust compiler and runtime
- TAL can eliminate compiler from trusted computing base
- Working on verifying the GC as well

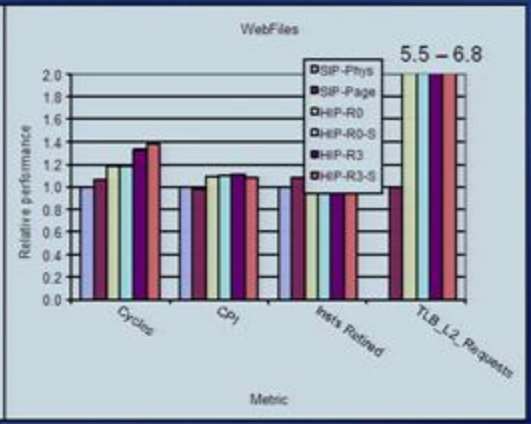


Hardware Protection Is Orthogonal



Cost Of Hardware And Software Isolation





Micro Benchmarks

Athlon64 3000+ (1.8GHz) nForce4 SLI	Cost (CPU Cycles)			
	Singularity	FreeBSD 5.3	Linux 2.6.11 (Red Hat FC4)	Windows XP (SP2)
Minimum kernel API call	**80**	878	437	627
Message request/reply	**1,041**	13,300	5,800	(LPC) 4,650 (NP) 6,340
Process create & start	**388,000**	1,030,000	719,000	5,380,000

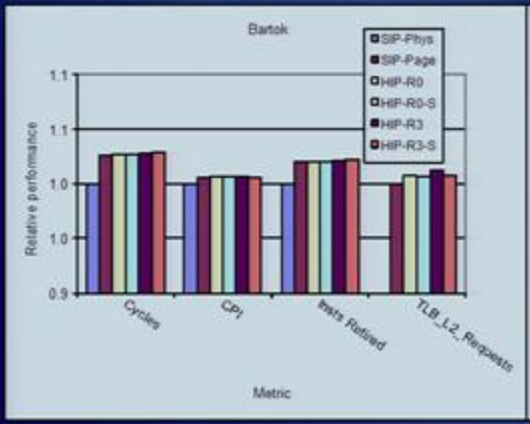
- Why?
  - all SIPs run in ring 0
  - static verification replaces hardware protection
  - good optimizing compiler (not JIT)

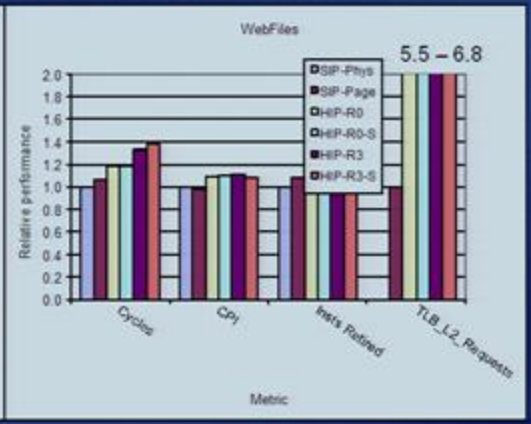
Challenge 3:

More Verification

- Integrate specifications throughout system
  - language
  - interprocess communication
  - system configuration
- Detect errors early, verify code late
  - language safety essential to system integrity

Cost Of Hardware And Software Isolation





Challenge 3:

More Verification

- Integrate specifications throughout system
  - language
  - interprocess communication
  - system configuration
- Detect errors early, verify code late
  - language safety essential to system integrity

Example:

Channel Contracts

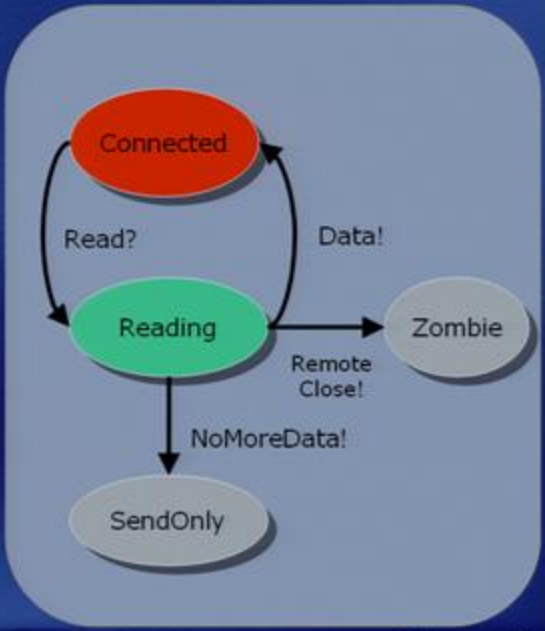
```
public contract TcpSocketContract {
...
state Connected : one {
    Read? -> ReadResult;
    Write? -> WriteResult;

    GetLocalAddress? -> IPAddress! ->
      Connected;
    GetLocalPort? -> Port! ->
      Connected;

    DoneSending? -> ReceiveOnly;
    DoneReceiving? -> SendOnly;
    Close? -> Closed;
    Abort? -> Closed;
}

state Reading : one {
    Data! -> Connected;
    NoMoreData! -> SendOnly;
    RemoteClose! -> Zombie;
}
...
}
```

? = receive
! = send



Example:

Channel Contracts

Contract

```
public contract TcpConnectionContract {
...
state Connected : one {
    Read? -> ReadResult;
    Write? -> WriteResult;

    GetLocalAddress? -> IPAddress! ->
      Connected;
    GetLocalPort? -> Port! -> Connected;

    DoneSending? -> ReceiveOnly;
    DoneReceiving? -> SendOnly;
    Close? -> Closed;
    Abort? -> Closed;
}

state Reading : one {
    Data! -> Connected;
    NoMoreData! -> SendOnly;
    RemoveClose! -> Zombie;
}
...
}
```

Client

```
...
conn.SendRead();
switch receive {
    case conn.Data(readData) :
        dataBuffer.AddToTail(readData);
        return true;

    case conn.RemoteClose() :
        return false;
}
...
```

Example:

Channel Contracts

Contract

```
public contract TcpConnectionContract {
...
state Connected : one {
    Read? -> ReadResult;
    Write? -> WriteResult;

    GetLocalAddress? -> IPAddress! ->
      Connected;
    GetLocalPort? -> Port! -> Connected;

    DoneSending? -> ReceiveOnly;
    DoneReceiving? -> SendOnly;
    Close? -> Closed;
    Abort? -> Closed;
}

state Reading : one {
    Data! -> Connected;
    NoMoreData! -> SendOnly;
    RemoteClose! -> Zombie;
}
...
}
```

Client

```
...
conn.SendRead();
switch receive {
    case conn.Data(readData) :
        dataBuffer.AddToTail(readData);
        return true;

    case conn.RemoteClose() :
        return false;
}
...
```

Missing Case

```
case conn.NoMoreData() :
```

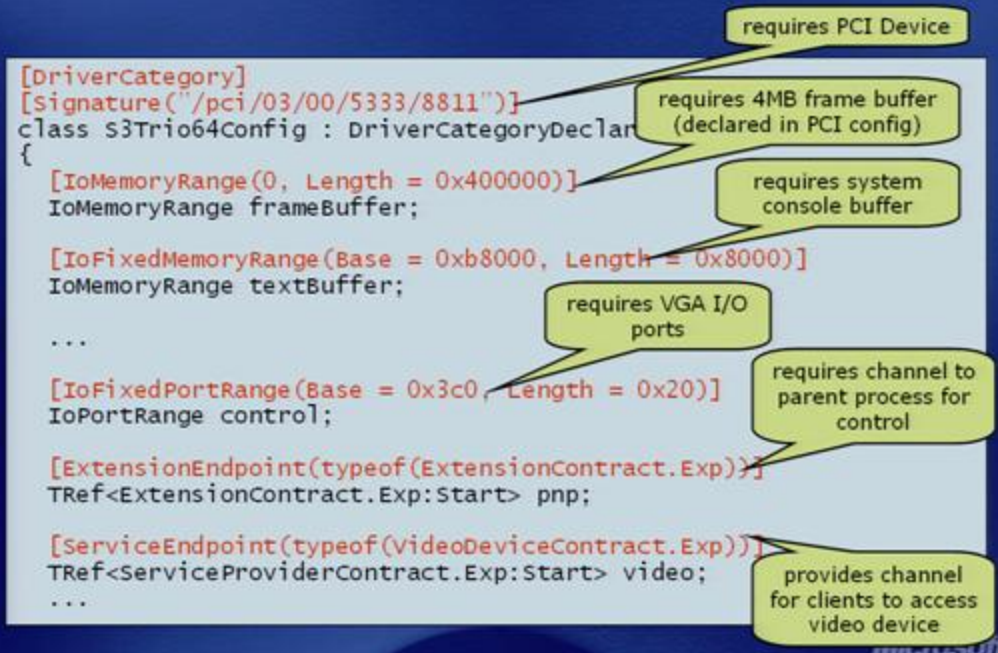
Contract conformance statically detects subtle errors such as deadlock

Example:

Applications Specifications

- Application is first-class abstraction with identity
  - code + resources + manifest
- Manifest specifies
  - software components
  - dependencies
  - exported channels
  - hardware or software resource requirements

Device Driver Specification



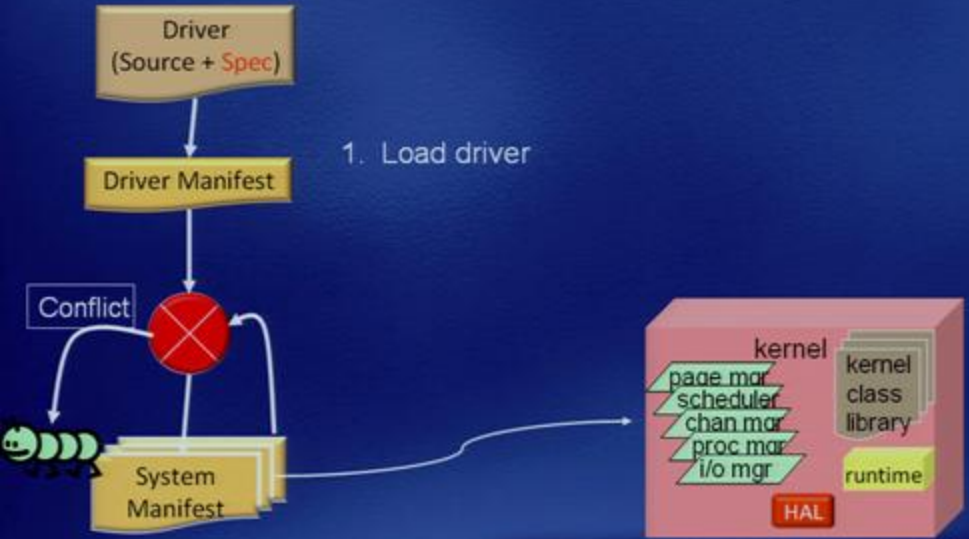
Specification Used In Many Ways

Specification Used In Many Ways

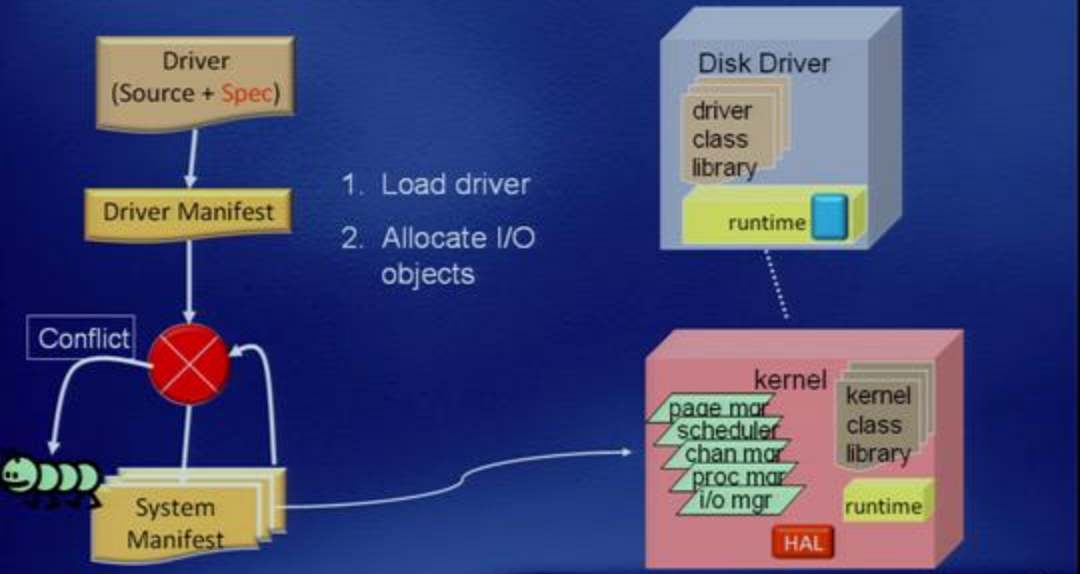
Specification Used In Many Ways

Specification Used In Many Ways

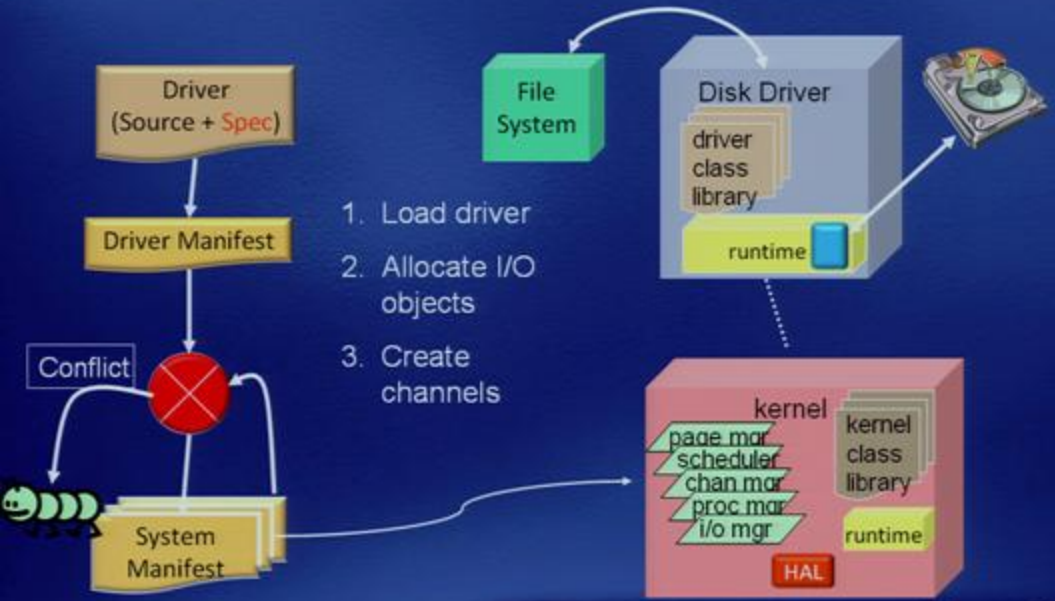
Specification Used In Many Ways



Specification Used In Many Ways



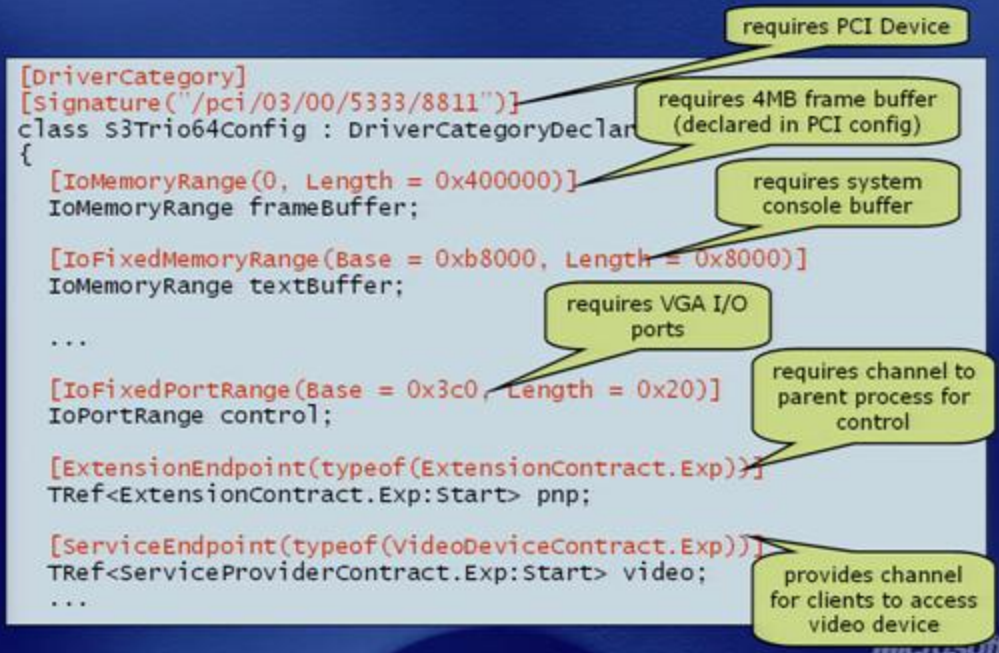
Specification Used In Many Ways



Verification Of System Configuration

- Verification ensures
  - never install an program that will break another program
  - never start a program without appropriate resources
  - never grant a program access to undeclared resources
- All of these checks performed statically

Device Driver Specification



Summary

- Singularity is basis for more dependable systems
  - pervasive use of safe programming languages
  - lightweight, closed, customizable run-time environment
  - verifiable specification of system behavior
- Working research prototype
  - driving research in large number of areas
- More information:
  - http://research.microsoft.com/os/singularity
  - Growing number of TRs & papers